ICS 53, Winter 2021

Assignment 4: A Memory Allocator

You will write a program which maintains a heap. Your program will allow a user to allocate memory, free memory, and see the current state of the heap. Your program will accept user commands and execute them.

I. Assumptions about the heap

The heap is 127 bytes long and memory is byte-addressable. The first address of the heap is address 0, so the last address of the heap is address 126. When we refer to a "pointer" in this assignment we mean an address in memory. All pointers should therefore be values between 0 and 126.

The heap will be organized as an implicit free list. The heap is initially completely unallocated, so it should contain a single free block which is as big as the entire heap. Memory is initialized so that all addresses (other than the header and the footer of the initial free block) contain 0. Each block should have a header and a footer which is a single byte, and the header and footer byte should be contained in memory, just before the payload of the block(header) and after payload of a block (footer). **The most-significant 7 bits of the header and footer should indicate the size of the block, including the header and footer itself. The least significant bit of the header and footer should indicate the allocation of the block: 0 for free, 1 for allocated**. The header for the first block (the initial single free block) must be placed at address 0 in memory and the footer of the first block must be placed at address 126.

You cannot assume that the blocks are aligned. This means that the start address of a block can be any address in the heap.

II. Operations

Your program should provide a prompt to the user (">") and accept the following commands. Your program should repeatedly accept commands until the user enters "quit". You only need to support these commands and can safely assume they will be entered and formatted in the manner shown below.

1. **malloc <int size>** - This operation allows the user to allocate a block of memory from your heap. This operation should take one argument, the number of bytes which the user wants in the payload of the allocated block. This operation should **print** out a pointer which is the **first address of the payload** of the allocated block.

   Example:

```
>malloc 10 // Comment: header at 0, payload from 1-10, footer at 11

1

>malloc 5 // Comment: header at 12, payload from 13-17, footer at 18

13

>malloc 2 // Comment: header at 19, payload from 20-21, footer at 22

20
```

2. **free <int index>**- This operation allows the user to free a block of memory. This operation takes one argument, the pointer to the start of the payload of a previously allocated block of memory. You can assume that the argument is a correct pointer to the payload of an allocated block.

   Example:

```
>malloc 10

1

>malloc 5

13

>free 13

>free 1
```

3. **blocklist** - This operation prints out information about all the blocks in your heap. The information about blocks should be printed in the order that the blocks are contained in the heap. The following information should be printed about each block: pointer to the **start of the payload**, **payload size**, and the **allocation status (allocated of free)**. All three items of information about a single block should be **printed on a single line and should be separated by commas**.

   Example:

```
>malloc 10

1

>malloc 5

13
```

```
>blocklist
1, 10, allocated.
13, 5, allocated.
20, 106, free.
```

4. **writemem <int index>, <char * str>** – This operation writes alpha-numeric characters into memory. The operation takes two arguments. The first argument is a pointer to the location in memory and the second argument is a sequence of alpha-numeric characters which will be written into memory, starting at the address indicated by the pointer. The first character will be written into the address indicated by the pointer, and each character thereafter will be written into the neighboring addresses sequentially. For example, the operation "`writemem 3 abc`" will write an 'a' into address 3, a 'b' into address '4', and a 'c' into address 5. Additionally, if a block is freed, you must ensure that whatever was written to the block is reset back to 0. **The index can be any location in the heap. Since C does not do explicit bounds-checking, the writmem command for your simulator can very well overwrite parts of another blocks even headers and footers and payloads of other allocated/free blocks. You are not required to do bounds checking for your heap as well. Operation like these will destroy the heap structure and are permitted operations. This can then corrupt subsequent operations as well.**

5. **printmem <int index>, <int number_of_characters_to_print>** – This operation prints out a segment of memory in **hexadecimal**. The operation takes two arguments. The first argument is a pointer to the first location in memory to print, and the second argument is an integer indicating how many addresses to print. The contents of all addresses will be printed on a single line and separated by a single space. **The index along with number of characters can very well exceed block sizes. Like writemem, you are not required to do bounds checking.**

   Example:

   ```
   >writemem 5 ABC
   >printmem 5 3
   41 42 43
   ```

   Notice that the values 41, 42, and 43 are the hexadecimal representations of the ASCII values of the characters 'A', 'B', and 'C'.

6. **quit** – This quits your program.

III. Requirements about allocation and freeing of memory.

When a block is requested which is smaller than any existing block in the heap, then **your code must perform splitting** to create a new block of the appropriate size.

When a block is freed, it must be coalesced with the next block if the next block is free and the previous block if the previous block is free. (Forward and Backward Coalescing needs to be supported by your code)

When searching for a block to allocate, be sure to use the **first-fit allocation** strategy.

IV. Developing Testcases

Testing is an important part of the programming process, so you are **required** to develop your own test cases. It is up to you to consider all the possibilities that can occur, within the limits of the specification, and make your test cases based on that. **We will grade your test cases based on coverage, i.e., you should come up with the test cases such that the whole suite of test cases is capable of 100% code coverage**, if you have a question about how the program is expected to perform, come to office hours and ask the professor, or go to lab and ask a TA. We have created a Python script called autocov.py which you should download with the assignment and use before you submit your assignment in order to ensure that you have achieved 100%-line coverage. The autocov.py script will compile and execute your code with a set of test cases which you define, and it will report the line coverage achieved by the test case. Do not modify the autocov.py script which we provide.

Test Runs

Testing is performed by executing your program several times with different test data. Each test execution of your code is referred to as a test run ("run" for short) and the test data associated with each run must be stored in a .run file. A .run file will contain all the test data which will be supplied to your program during a single test execution. A .run file is a text file describing all the inputs supplied to your program for a single execution. The name of each .run file must have the suffix ".run", but the prefix does not matter. Each line of a .run file can be no longer than 80 characters.

Input to autocov.py.

The autocov.py script requires your access to the source code of your program and the set of .run files to be used during testing. Your source code file must be named "hw.c". This is important; it is the only file which will be compiled by the script. Any number of .run files may be provided. All these files, hw.c and all of your .run files, must be "zipped" together into a single zip file named "hw.zip". You can create the zip file using the "zip" command on linux. For example, if you have two .run files called "t1.run" and "t2.run" then you could create the zip file with the following command: "zip hw.zip hw.c t1.run t2.run". The hw.zip file is the only input to the autocov.py script.

Running autocov.py

In order to execute the script, the autocov.py file must be placed in the same directory as the hw.zip file and another directory called “hw”. The hw directory will be used by the script to compile and run your program, so it should be empty before running the script. The script needs to be run using a Python interpreter version 3.6.8 or later. On the openlab systems the command “python3” invokes the Python 3.x interpreter, so you can run the autocov.py script by entering the command “python3 autocov.py”. The script will print several lines to the screen, including program outputs, but the second-to-last line should look like this, “Lines executed: X% of Y”, where X is the line coverage and Y is the number of lines in your program.

V. Submission Instructions:

Your source code must be a single c file named ‘hw.c’ containing your solution and any number of run files containing your suite of test cases zipped into 1 file named ‘hw.zip’. Be sure that your program must compile on openlab.ics.uci.edu using gcc version 4.8.5 with no compiler switches other than gcov switches.

Submissions will be done through Gradescope. If you are working with a partner, you must indicate that through Gradescope. The first line of your submitted file should be a comment which includes the name and ID number of you and your partner (if you are working with a partner).

VI. Sample Execution

```
>malloc 10
1
>malloc 5
13
>blocklist
1, 10, allocated.
13, 5, allocated.
20, 106, free.
```

>free 1

>blocklist

1, 10, free.

13, 5, allocated.

20, 106, free.

>malloc 5

1

>blocklist

1, 5, allocated.

8, 3, free.

13, 5, allocated.

20, 106, free.

>writemem 1 HELLO

>printmem 1 5

48 45 4C 4C 4F

>free 13

>blocklist **// Comment: You can see here that there was a block allocated between 2 free blocks, on freeing that allocated block, both the free blocks were coalesced into one free block.**

1, 5, allocated.

8, 118, free.

>quit

$ <- back to bash prompt.